GPU-based Real-Time Collision Detection for Motion Execution in Mobile Manipulation Planning

Andreas Hermann, Sebastian Klemm, Zhixing Xue, Arne Roennau and Rüdiger Dillmann

***Abstract*—In this paper we present a parallel collision checking approach as an essential building block of a reactive online planning framework, that allows to continuously monitor the execution of planned trajectories against dynamic changes in the environment. The software is optimized for massively parallel hardware architectures, namely CUDA GPUs and offers constant runtime regardless of the occupancy density in the environment.**

I. Introduction

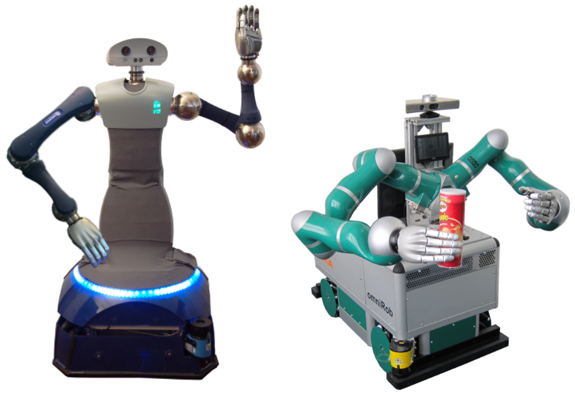
A. Motivation

Mobile Manipulation is a key functionality for the practical usage of service robots in domestic as well as in industrial environments. But it also holds a lot of difficulties for an autonomous system: Apart from the unstructured environment and the resulting perception challenges a robot has to deal with permanent changes in its surroundings. Created plans for motions turn invalid within seconds if new obstacles appear in the proximity of the robot. This requires a continuous validation of plans and a fast replanning in case of new constraints. By interweaving planning and execution, a reactive behaviour can be achieved. That again requires a fast collision detection for two reasons: For motion planning in general (planners spend 90 percent and more of calculation time with collision-checking [1]) and for a real time monitoring of the motion execution to stop the robot in time before an unavoidable crash.

Today mesh-based collision checkers that consider 3D depth data are mainly used in planners for rather coarse sense-plan-act cycles and not for real time monitoring of the execution of planned trajectories. This results from the computation complexity: The necessary triangulation of point clouds to meshes as a preprocessing takes time. Also the actual runtime for collision checking depends on a multitude of conditions, like the level of geometric complexity of the robot model and the degree of occupancy and clutteredness of the environment. Therefore live 3D depth data is currently mostly used to check collisions only for large areas as a whole, e.g. safety margins around a mobile platform. In contrast to that, our specialized parallel approach is independent of the described metrics and runs with guaranteed response times. Execution on a many-core GPU is fast enough to allow continuous fine-grained monitoring of the robot's motions while it is still possible to query further checks for reactive motion planning simultaneously.

B. Application scenario

Our application scenario is motion planning in dynamic real world environments for the highly articulated bimanual mobile manipulators shown in Fig. 1. The robots are equipped with one or more RGBD-Sensors to sense their environment. HoLLiE [2] shown in Fig. 1(a) features an active upper body that allows her to pick up items from the floor. The Robot IMMP (Fig. 1(b)) [3] has long arms to place and transport objects on the rear side of his platform. Different upper body configurations result in highly variable geometries for both robots, requiring a full 3D planning model even when only their mobile platforms are operated.



(a) HoLLiE (b) IMMP

Fig. 1: The highly articulated mobile manipulation hardware platforms HoLLiE and IMMP that evaluate their motions with the presented collision checker.

This paper focuses on the evaluation of motions with regards to collision detection. We describe a high performance real time collision checker, based on superimposing two voxelmaps that are maintained on a Graphics Processing Unit (GPU). The remainder of this paper is organized as follows: First we set our approach into the context of related work. Data structures and map maintenance for the collision detection are detailed in Sec. III, followed by an application scenario in Sec. IV. The experiments profiling our algorithms are listed in Sec. V. Finally, we compare our results to other work and present an upcoming planner in the outlook.

II. Related Work

Robotics motion planning made huge progress during the last years. Static planners with the requirement for a fully

The research leading to these results has received funding from the German Federal Ministry of Education and Research (BMBF) under grant agreement no. 01IM12006C (ISABEL) and from the Federal Ministry of Economics and Technology (BMWi) under grant agreement no. 01MA09049A (viEMA).

All authors are with the department of Intelligent Systems and Production Engineering (ISPE - IDS) Research Center for Information Technology (FZI), 76131 Karlsruhe, Germany {hermann,klemm,xue,roennau,dillmann}@fzi.de

978-1-4799-2722-7/13/$31.00 © 2013 IEEE

a priori known environment model are replaced by dynamic approaches that try to cope with highly complex problems in an online way. Sampling based planners use direct kinematics to transfer poses from Configuration ($\mathcal{C}$)-Space into the Work ($\mathcal{W}$)-Space to perform collision tests and to evaluate the feasibility of a motion trajectory. These checks can be based on triangle-mesh-models like in computer graphics or work with cell decomposition that discretizes the work space into occupied or free cells. Both approaches have their special up and downsides, considering not only computation complexity:

Planners which require an exact geometric representation like grasp planners [4] or physics simulations are triangle-mesh based to be able to determine contact surfaces. The meshes can be organized in hierarchical structures and processed on a GPU [1] or surrounded by bounding volumes that allow collision tests and distance measurements without working directly on triangles [5]. The runtime of mesh based approaches is heavily dependent on many boundary conditions as described above. The heuristics for automatic decomposition of huge or complex meshes can cause high runtimes. Also the intersection check between a measurement point clouds and a meshes requires the triangulation of the point cloud which has a high cost [6] of $\mathcal{O}(n^3)$ for Delaunay triangulation of an unordered cloud or at least $\mathcal{O}(n)$ for an ordered cloud of size n. A combined library for collision-detection and distance-calculation between meshes and point-clouds is FCL [7]. The according paper also gives a good overview on the whole topic.

Our implementation uses well known cell decomposition methods, for which all these problems are not given, as abstraction and collision checking can inherently happen in the underlying data structure. 3D measurement data is directly used to update the probabilistic map through a Bayesian process [8]. That also makes the integration over time, the subsampling and the merging of diverse data sources very convenient and achievable in constant time bounds. Collision checking becomes a bulk of independent comparison operations that can be executed in parallel. A CPU based Voxel collision check exists in the ROS Collider Package [9].

Data structures to store maps of cells can be optimized and ordered in a multi-resolution way to optimize occupancy queries. Popular structures for 2D and 3D data are trees: Octrees combined with artificial potential fields in Voxelspace have already been used back in 1995 [10]. Behnke [11] showed another kind of multi-resolution maps as he decreases the gridmap resolution in accordance to sensor uncertainty in rising distance and planning time horizon. Such hierarchical structures can be used with our approach as robot centric map.

Calculation times for finding feasible trajectories can not only be reduced through simplification of the maps but also by spatial or temporal abstraction of the robot model: The work of Vahrenkamp [12] dynamically disables robots DOF to reduce $\mathcal{C}$-Space dimensionality. Another possibility to disregard active DOFs while keeping their movements collision free is to cover a whole motion range with a $\mathcal{W}$-Subspace: The generation of so called swept volumes on a mesh basis is a computationally intensive process, as shown in [13]. Nevertheless [14] implemented a real time capable self collision avoidance based on swept volume approximations. As our generation of swept volumes is Voxel-based, it does not suffer from high computational efforts.

All state of the art planning approaches that use parallel collision querying can gain a runtime improvement by using our approach. Some example strategies are listed here:

- For problems with high-dimensional $\mathcal{C}$-Space like Kinodynamic planning, replanning can be used to adjust only critical sections. Already generated plans are reused, like in [15]. As described later, we support this through storing swept volumes of sub-plans.
- Roadmap based planners build up a graph where edges represent motions [16]. During runtime edges may be invalidated by obstacles but alternative routes can be found efficiently by parsing the abstract graph. The maintenance of such a graph is easy to achieve with our software as it can evaluate many paths in parallel without runtime increases.
- Another way to optimize search space is to implement hierarchical approaches, where a coarse planner constrains one or more local planners. The works of [17] and [18] give a good overview of such algorithms. Our collision checker supports the required multiple simultaneous queries.

III. Collision Detection

An overview of our software is shown in Fig. 2(a): Two 3D Voxelmaps hold discretized representations of the environment data respectively the robot plus its swept volumes. For a collision check, these maps are superimposed. The contribution of our work lies in the data structures and algorithms that are described in this section, which are optimized for highly parallel execution on the GPU.

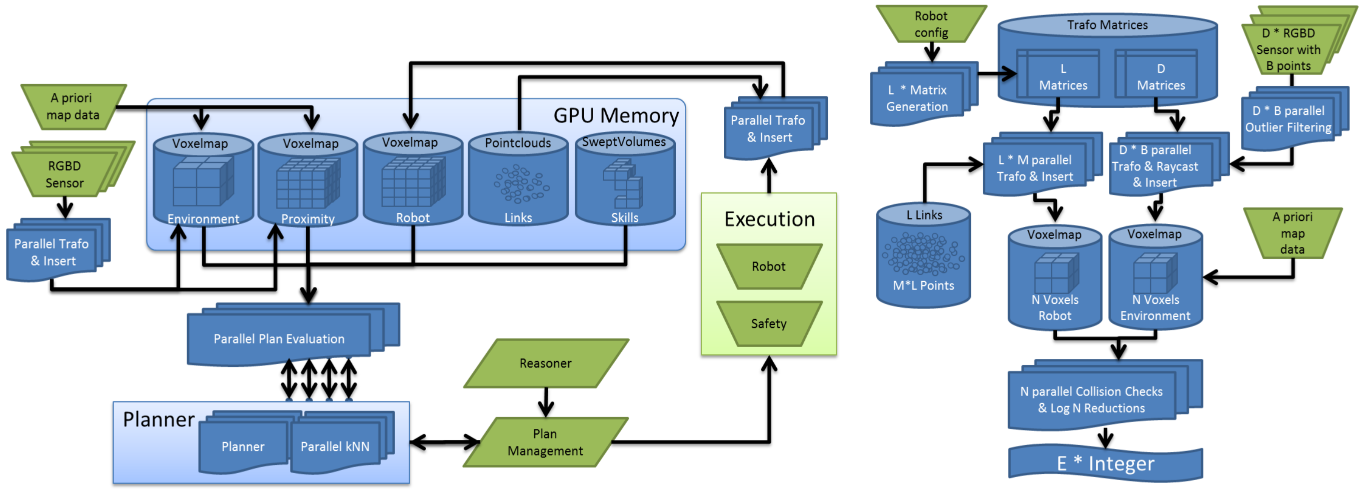
A. *Implementation influencing considerations*

Our approach basically works with all kinds of data structures that allow a random access within $\mathcal{O}(1)$ and which can decompose the environment and the robot into discrete geometric cells. In the two-dimensional case, the preferred structure would be an array of hexagon or square patterns.

As memory management and the allocation of new objects in dynamic data structures compromises constant runtimes and conflicts with the GPU programming paradigmes, the usage of a memory-efficient space partitioning tree as map data structure was not a viable option. Instead we use a Voxelmap with an alternative space decomposing addressing scheme (Morton-Z-Ordering [19]) to quickly calculate the memory address of a node, its neighbours, children and parent. This is useful for planning scenarios that require different levels of detail.

B. *Voxelmap*

Our Voxelmap stores cubic Voxels in a one dimensional array in GPU memory. Addressing of the cells is optimized for clustered read/write access on the GPU. Each cell holds a probability indicating the cell's occupancy together with a key identifying its meaning. In the environment map a Voxel may be of the following kind: Unknown, static a priori map data, dynamic sensor data, ray cast free space. As a Voxel can only hold one occupancy but may be associated with more than one



(a) Overview over the software architecture.

(b) Details of the parallelization of our collision check.

Fig. 2: Flowcharts of the implementation. Components in blue are running on the GPU.

meaning, a list of simple rules manages the update process. E.g. a formerly free Voxel changes its type if it is detected occupied. If a priori static map data is seen by the sensors it is identified as such but its 100% occupancy probability remains untouched. Voxels in the robot map can belong to the real robot or to one or more swept volumes. No special update rules are needed here.

To map geometric positions $p = (x,y,z)^T, x,y,z \in \mathbb{R}$ to the memory addresses of according Voxels (VA) we use

$$VA(x,y,z) = B + (\lfloor \frac{z}{VS_z} \cdot MD_x \cdot MD_y \rfloor + \lfloor \frac{y}{VS_y} \cdot MD_x) \rfloor + \lfloor \frac{x}{VS_x} \rfloor)$$

where B is a constant memory offset calculated at program launch, VS reflects the size of the voxel in each dimension and MD holds the dimensionality of the voxelmap.

We implemented a Voxelmap (see Fig. 3) with arbitrary geometry that requires minimal memory and offers unbounded parallel access while also fulfilling all requirements for the parallel algorithms that are described in the following sections.

C. Point Cloud data insertion

As there are a lot of different depth sensors with different resolutions for particular purposes it is important to be able to integrate their data into a common map. This works well together with the used Voxel grids, which also make it superfluous to tessellate point clouds before collision checking can be realized.

The following processing pipeline can be executed in parallel with up to one thread per measurement point (optimum thread-count is discussed in Subsec. V-A): Acquired point clouds are raw-copied into GPU memory, where a statistical outlier filtering is done, classifying points by their euclidean distance to their neighbors. The reliability of the remaining points is estimated according to a sensor model before they are transformed into the global coordinate system (see top right part of Fig. 2(b)). After transformation the coordinates of each point are discretized to determine the according Voxel, whose occupancy status is updated as a Bayesian process. During insertion of points into the environment map corresponding Voxels' meanings in the robot map are reviewed if they are occupied by the robot model. This prevents the insertion of measurement points originating from robot parts. Finally, free space is determined by executing a ray casting which runs in one thread per ray that is send from the sensor pose towards a measurement point. We use a generalized version of the well-known Bresenham algorithm for 3D-space that updates the visited Voxels' occupancy according to the sensor model.

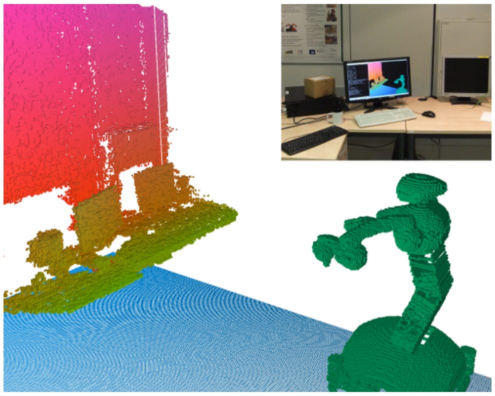


Fig. 3: Visualization of the dynamic Voxelmaps used for collision checking: Robot map and environment data of a scene seen by the robot

D. Voxelization of the robot model and Swept volume generation

Given a detailed CAD surface mesh model of our robots, we generate point clouds of each part of the kinematic chain in an offline process: The single rigid robot parts are sent through a modified ray casting algorithm from [20] to generate a dense volumetric point cloud of it. Contrary to a regular raycast which only generates points on the surface of the input

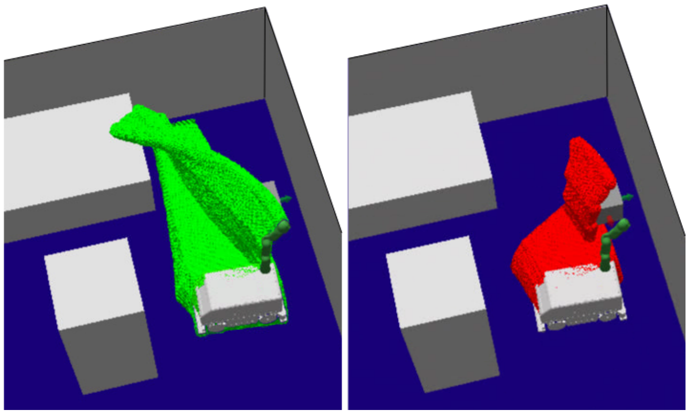
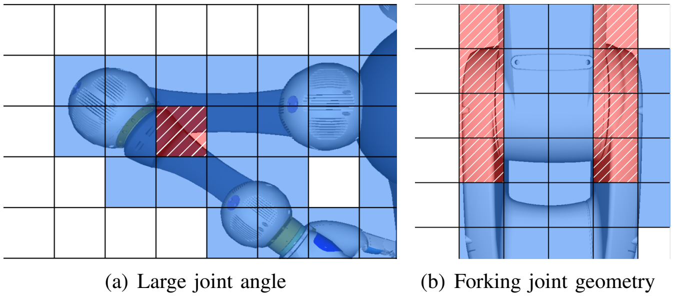


Fig. 4: Voxelized Swept volume generated by planned motion (green) for the robot IMMP. Parts of the swept volume in collision (red) when obstacle box is moved.

mesh, our computation also generates equidistant points within objects. The resulting point clouds are loaded at program start and are kept read-only in GPU memory. During runtime they are copied and each segment is transformed by DH-parameters of a linked kinematic chain (see top left part of Fig. 2(b)). The transformed point clouds are finally inserted into the robot map. By always using the initial clouds and not their Voxel representation for transformation, the discretization error is minimized. The transformation and Voxelization is done in the same parallel way as with the measurement data, with up to one thread per point.

By keeping former poses in the map and assigning them new IDs instead of clearing them, a swept volume of a motion trajectory can be generated in the robot map. This is described in Subsec. IV-A.



(a) Large joint angle

(b) Forking joint geometry

Fig. 5: Schematic visualization of a misleading self-collision that cannot occur when considering kinematic constraints. Due to our modelling no such false positives are detected.

Self-collisions can be detected by inserting single robot parts into the Voxelmap in a predefined and alternating order paying attention to the robot's kinematics. That way no false collisions appear when handling intersecting links or extreme joint angles, as shown in Fig. 5. Mechanical joint constraints prevent subsequent joints from self-collision inherently and have not to be checked. Whenever else the insertion procedure discovers that a robot part would overlay an already occupied Voxel, a self-collision is discovered.

E. Massively parallel collision detection

The development aims at significantly reducing the cost of collision checks to allow the simultaneous execution of a high number (> 100) of checks and therefore supports new planning approaches. By working directly on Voxel representations of both the robot and the environment, our approach benefits from the absence of time consuming mesh generation and can handle live sensor data with only little preprocessing.

Our collision detection superimposes two Voxelmaps of the same geometric dimension and resolution and checks if both possess occupied nodes at the same position (see middle and bottom part of Fig. 2(b)). The occupancy certainties of corresponding Voxels are checked against thresholds to decide if a collision is present. Following CUDA's SIMT[1] programming paradigm this is done with a single Kernel[2] call, using as many threads as there are Voxels in the map up to the CUDA grid size $g = \hat{t} \cdot \hat{b}$, where $\hat{t}$ is the maximum number of threads per block and $\hat{b}$ the number of blocks on the CUDA grid (for our hardware $g = 67.10784 \cdot 10^6$). The underlying framework then schedules all threads to the hardware. Voxelmaps that hold more than g Voxels are handled by shifting Voxel addresses by g and iteratively processing remaining Voxels.

But as thread management comes at a cost, it is reasonable to perform more than one collision check per Kernel call, meaning that each running Kernel loops and handles l Voxels, $l > 1$. This can be traced back to depend on the number of processors on the GPU, the number of CUDA blocks used and to caching effects. The influence of this factor is also shown in the experiments section. Looping l-times within a Kernel of course leads to a factor l larger number of Voxels that can be assigned to CUDA grid threads at once.

The result of each collision check Kernel call is an integer, indicating how many Voxels collided. To minimize Memcopy overhead for final evaluation on the CPU, the Kernel performs a reduction for each CUDA block, summing up the results of all according threads, which leaves an array of size b (b : number of used blocks). This array is copied to the host and summed up again. The final number of collisions gives an impression of the collision severity, that can later be used to influence the planer.

Key advantages of our techniques are that the time bounds on the collision check strictly depend only on the size of the Voxelmap and are independent of the number of measurement points, the degree of occupancy of the environment or the complexity of the robot model. Further, due to the Voxel meanings described earlier, it is possible to evaluate M robot poses or whole subsets of plans in one check, where M is only limited by available GPU-RAM. In other words the check of M future plan samples runs in the same constant time, as a single pose check. This is a huge advance compared to collision checkers that check each of the M samples in a series.

The massive parallelization on the CUDA hardware is the key for reactive planning and offers the possibility of parallel

[1] Single Instruction Multiple Threads

[2] A Kernel is a function that runs with many parallel threads on the GPU which is launched by the host.

investigation of different plans at no extra computational cost while the CPU can be used for further tasks.

IV. Application in Opportunistic Reactive Parallel Planning

The general intention for developing a parallel collision checker was to allow faster and parallel planning to enable a reactive motion generation in dynamic and partly unknown environments. One key feature to achieve this is the creation of alternative plans and the continuous evaluation of their feasibility. If one plan gets invalidated during execution time, a global goal can still be achieved by switching to another plan seamlessly or by replanning small segments on the fly.

One example could be the use of a planner as shown in the work of [21]. It lists three different approaches on parallelizing the A* algorithm.

A. Evaluation of poses and swept volume creation

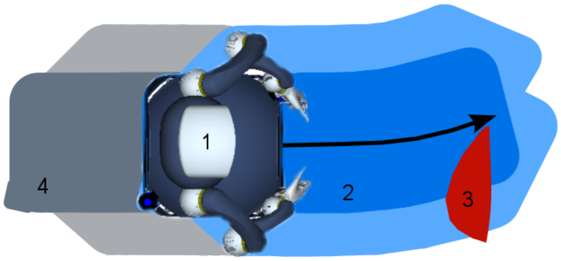


Fig. 7: Sketch of temporal forward projected swept volume of HoLLiE, seen from top. 1) Current pose, 2) Swept volume, 3) Obstacle, 4) Removed volume of completed motion

When a transition is queried by a planner, it is first evaluated against the current environment (top row in Fig. 6). This phase can be considered as a virtual robot preceding the real robot on the intended transition, while a swept volume of the motions in the $\mathcal{W}$-Space is recorded, like the one in Fig. 4. If the virtual forerunner gets stuck before execution starts, the planner is informed to query an alternative transition. As soon as a section with a specific length is checked, the real robot follows with the execution, moving within the safe corridor formed by the temporal forward projected swept volume of the virtual robot (bottom row in Fig. 6). This volume is continuously collision checked with live sensor data to detect new obstacles that protrude into the covered space, as illustrated in Fig. 7. If such a situation is detected early enough, a local replanning is triggered around the point of the collision. If an alternative sub-plan is generated fast enough it can replace the old one and execution is continued without interrupting the motion.

Once generated swept volumes can be annotated and stored to build a library of basis operations. With that, it is possible to quickly evaluate feasible locations to execute an operation without detailed motion planning. Our collision checker offers the required features to evaluate different poses or swept volumes of whole subplans at the cost of a single check.

V. Experiments

In this section we describe our approach to optimize GPU throughput, evaluate single processing steps and quantify the performance of our algorithm.

A. Empirical analysis of threads per task impact

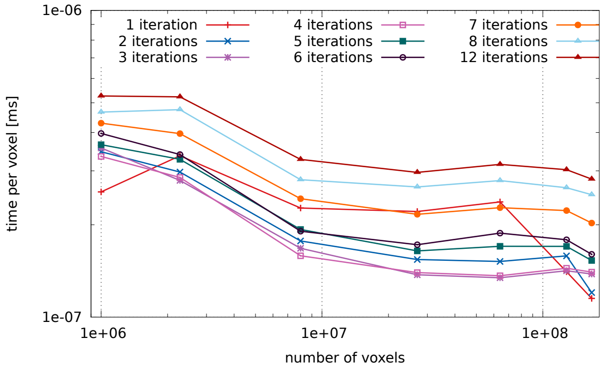


Fig. 8: The collision check processing time per Voxel shows the impact of different parallelization factors on GPU throughput. Both axes are drawn in logarithmic scale.

The most important restrictions in GPU programming are the complexity and controversy in handling dynamic memory allocations from within Kernels and the optimized execution of data processing in a pipeline. This complicates dynamic programming patterns. Regarding that, the efficiency of a highly parallel program on the GPU is best, when it is designed as a pipeline, with all steps running on the GPU. Especially when all data is continuously kept on the device's memory, the throughput can be maximized as no memory interaction from the CPU is needed. We followed that principle in our implementation.

A variable that is crucial for the throughput of the GPU is the factor that determines the relation of threads per task, defined by l introduced in Subsec. III-E. To assign this factor, one has to consider that the CUDA runtime environment schedules the assignment of the working threads to the cores. This works best if enough threads are available. On the other hand, the overhead of a context switch on a GPU multiprocessor implies a time overhead. Threads are further grouped in blocks, so it is not possible to calculate the perfect factor in a straight forward way, especially as block boundaries are not met exactly in most of the cases. The warp size (the number of physically simultaneously executed threads on a multiprocessor), the processor's cache line size, the number of used blocks and the alignment of data in global memory are values to be considered when determining the optimal number of loops per thread vs. a mostly-wide parallelization of the load. The empirical determination of this factor for collision checks is plotted in figure 8. As we see, the impact changes with varying map size.

B. Analysis of algorithm complexity and hardware specific factors

Our hardware consists of a NVidia Titan GTX with 6 GB GDDR5 frame buffer memory and a bandwidth of 288.4

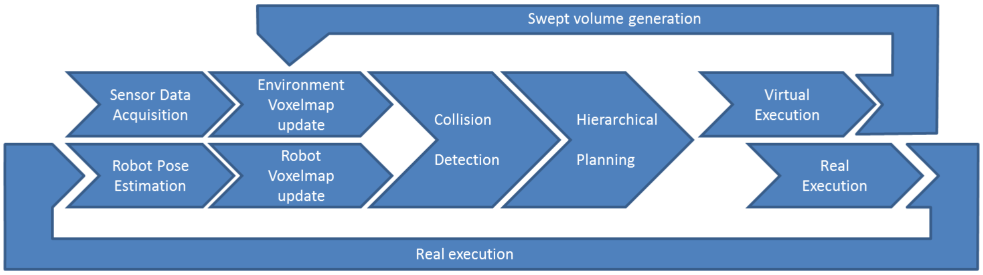


Fig. 6: Main program loop showing interweaved continuous planning and execution. Data acquisition is done continuously.

GB/s. The GPU combines 2688 CUDA processing units with a warp size of 32 threads each. Onto that hardware the CUDA runtime assigns $65\,535 (= 2^{16} - 1)$ blocks, each capable of handling 1024 threads. That makes a total amount of 2^{26} parallel launchable threads, involving a physically parallel execution of 86016 threads at a time.

A collision check Kernel invokes a set of operations for which the runtime complexity can be calculated as follows (n: number of Voxels, $\hat{t}$: number of maximum possible threads on the grid, b: number of blocks used, t_b: number of threads per block used, l: number of loops within a thread):

- Calculation of the required number of threads and time consumption:
  $n/l \leq \hat{t}$: 1 division, 1 addition, 2 assignments, constant time
  $n/l > \hat{t}$: 2 assignments, constant time
- Execution of the Kernel: depending on the number of Voxels and therefore number of needed threads
  $n/l \leq \hat{t}$: 1 comparisons, constant time
  $n/l > \hat{t}$: $\lceil \frac{n/l}{k} \rceil$ comparisons, linear time depending on n
  reduction: $r := \log_2 t_b$ steps which runs in a bounded amount of time, as t_b is capped ($t_b \leq 1024$), so $r \leq 10$.
- Copy resulting data: the collision information of each block has to be copied to the host. This equals a memcopy of 5 Byte per block (for $l = 4$ loops within a Kernel) and is bounded by 327 675 Byte.
- Processing of the copied data on the host side: for evaluation of the $\lceil b/l \rceil$ values $\lceil b/l \rceil - 1$ additions and one comparison have to be performed: linear time consumption depending on b.

In most of our use-cases, the maps have less than $3 \cdot 2^{26}$ Voxel, so we benefit from being able to address all Voxels within the CUDA grid boundaries. Therefore we achieve an optimal runtime without relaunching the memory shifted kernel.

C. Program profiling

Average timings for inserting Kinect measurements with 307 200 points are as follows: The Memcopy to transfer the data from host to GPU RAM takes 0.7108 ms in average. The transformation is a lot faster with only 0.2075 ms. Determining the according Voxels and updating their occupancy takes another 0.065 ms, making a total of 0.9833 ms regardless of the map size in which we insert, as the addressing of a Voxel is always accomplished in a constant amount of time. As a comparison, the acquisition of triangulated Kinect data took 47.76ms per frame in average using PCLs OrganizedFastMesh (Edge lenght = 1 px) in our tests.

As described earlier, the point clouds representing the robot's links are stored on the GPU and only need to be transformed and inserted into the robot map. For a robot model with 9 kinematic links and a total of 188 999 points this takes 0.936 ms. As the same procedure is used for the generation of swept volumes like the one in Fig. 4, that generation happens at almost no extra cost.

After the environment map is populated with measurement point clouds and the robot map holds the swept volume and the current robot configuration, the collision check can be run. The runtime of this step depends on the Voxel map size. Example times and memory consumption for different map sizes are listed in Table I. For a mapsize of 400^3 the summed up response time from sensing an obstacle until signalling a collision is 9.58 ms, which leaves sufficient time to safely stop our robot's motion in an emergency situation.

			time [ms]	
Voxelmap dim	# of Voxels	size [MB]	for $l = 1$	for $l = 3$
$100 \times 100 \times 100$	1 000 000	15.2588	0.256	0.356
$150 \times 150 \times 150$	3 375 000	51.4984	0.764	0.635
$200 \times 200 \times 200$	8 000 000	122.0703	1.815	1.341
$300 \times 300 \times 300$	27 000 000	411.9873	5.967	3.711
$400 \times 400 \times 400$	64 000 000	976.5625	15.193	8.599
$400 \times 400 \times 800$	128 000 000	1953.1250	17.997	18.126
$550 \times 550 \times 550$	166 375 000	2538.6810	19.149	22.998

TABLE I: Combined GPU memory consumption for two independent maps of equal dimension and collision checking times for the non-optimized ($l = 1$) and for the for our use case optimized ($l = 3$) Kernel calls (average over 10 000 measurements). Includes reduction within Kernel, Memcopy to and evaluation on host.

The listed times show, that our approach can be used for real-time collision checking: For a Voxelmap of 550^3 Voxels there is theoretically still enough time to cope with up to 25 RGBD cameras in parallel, updating the Voxelmap with measurement data at 30 Hz. As mentioned before, each collision check can validate a set of c configurations at once, where c is only limited by available GPU memory as it influences the Voxelmaps' storage size, further reducing the effective duration of a collision check for a single configuration. E.g. on a $400 \times 400 \times 400$ dimensioned Voxelmap the effective collision time per configuration reduces to 0.0423 ms querying $c = 200$ configurations at once.

It is hard to compare these results to the related work, as our approach can handle a high number of collision checks within one run-cycle independently from the scene's geometric shape complexity the robot is situated in, the robot's own geometric shape complexity, and the number of objects within the scene that have to be considered for collision checking—while the runtime of mesh-based approaches is heavily dependent on these boundary conditions, regarding the tested meshes. When comparing the runtimes of map updates, it has to be considered that we deal with a fixed map size, while Octree implementations can handle growing map sizes.

Nevertheless the time for map-updating that we presented is a lot shorter than in CPU-based Octree mapping. E.g. OctoMap inserts point cloud data with around 300 ms for 300.000 points into a 10 cm grid. Collision detection in the OctoMap based ROS Collider Package runs with 2-4 Hz in typical indoor scenes with less dense point clouds and larger grid size [22].

Triangle based collision detection on the GPU can currently achieve a throughput of 21k triangles per 1 ms when using a run-time optimized data structure with high memory-consumption [23]. Typical indoor scenes easily have more than 1000k triangles, why collision detection is commonly done on bounding volumes. For narrow passages still a time consuming detailed check is required. Runtime grows linearly with the number of poses to check. This is why our method outperforms triangle based methods when several checks are queried at once. As a confirmation we ran self-collision detection on our CAD model of HoLLiE through the ROS FCL Library [7] which needed 4.769 ms in average over 300 checks.

VI. Conclusions and Future Work

A. Conclusions

A fast collision detection is needed for online motion planning. Especially when different planner instances run in parallel next to live collision checks with an operating robot. Therefore we implemented a real-time capable collision checker with strict runtime boundaries, that are independent from the amount of inserted data. The map structures are optimized for interweaved execution and planning as they can hold and check live sensor data against a multitude of poses and swept volumes of whole upcoming plans. Our GPU approach for the collision checking offers a 1-2 orders of magnitude higher performance compared to CPU based algorithms.

B. Outlook

Follow up work will describe our multilevel planner implementation that exhaustively uses parallel collision checks to speed up mobile manipulation planning and allows a fast adaptation in the case of collisions. Also we want to evaluate the dynamic memory and Kernel capabilities of the Kepler GPU architecture to work with Octrees instead of Voxelmaps.

References

[1] J. Pan, C. Lauterbach, and D. Manocha, "g-planner: Real-time motion planning and global navigation using GPUs," in *AAAI Conference on Artificial Intelligence (AAAI)*, 2010.

[2] A. Hermann, J. Sun, X. Zhixing, S. Ruehl, J. Oberlaender, A. Roennau, J. Zoellner, and R. Dillmann, "Hardware and Software Architecture of the Bimanual Mobile Manipulation Robot HoLLiE and its Actuated Upper Body," in *Advanced Intelligent Mechatronics (AIM), 2013 IEEE/ASME International Conference on*, jul. 2013, pp. 286 –292.

[3] A. Hermann, Z. Xue, S. Ruehl, and R. Dillmann, "Hardware and software architecture of a bimanual mobile manipulator for industrial application," in *Robotics and Biomimetics (ROBIO), 2011 IEEE International Conference on*, dec. 2011, pp. 2282 –2288.

[4] Z. Xue, P. Woerner, J. M. Zoellner, and R. Dillmann, "Efficient grasp planning using continuous collision detection," in *IEEE International Conference on Mechatronics and Automation (ICMA)*, 9-12 Aug. 2009, pp. 2752–2758.

[5] M. Saha, G. Sanchez, and J. Latombe, "Planning multi-goal tours for robot arms," in *International Conference on Robotics and Automation*, Taipei, Taiwan, 2003.

[6] P. Cignoni, "DeWall: A fast divide and conquer Delaunay triangulation algorithm in Ed," *Computer-Aided Design*, vol. 30, no. 5, pp. 333–341, Apr 1998.

[7] J. Pan, S. Chitta, and D. Manocha, "Fcl: A general purpose library for collision and proximity queries," in *Robotics and Automation (ICRA), 2012 IEEE International Conference on*, 2012, pp. 3859–3866.

[8] A. Hornung, K. M. Wurm, M. Bennewitz, C. Stachniss, and W. Burgard, "OctoMap: An efficient probabilistic 3D mapping framework based on octrees," *Autonomous Robots*, 2013.

[9] A. Harmat, G. E. Jones, K. M. Wurm, and A. Hornung, "ROS Collider Package, http://ros.org/wiki/collider."

[10] Y. Kitamura, T. Tanaka, F. Kishino, and M. Yachida, "3-d path planning in a dynamic environment using an octree and an artificial potential field," in *Intelligent Robots and Systems 95. 'Human Robot Interaction and Cooperative Robots', Proceedings. 1995 IEEE/RSJ International Conference on*, vol. 2, aug 1995, pp. 474 –481 vol.2.

[11] S. Behnke, "Local Multiresolution Path Planning," *In Proceedings of 7th RoboCup International Symposium*, pp. 332–343, 2003.

[12] N. Vahrenkamp, E. Kuhn, T. Asfour, and R. Dillmann, "Planning Multi-Robot Grasping Motions," *wwwiaim.ira.uka.de*, 2010.

[13] Y. J. Kim, G. Varadhan, M. C. Lin, and D. Manocha, "Fast swept volume approximation of complex polyhedral models," *Computer-Aided Design*, vol. 36, no. 11, pp. 1013 – 1027, 2004.

[14] H. Taubig, B. Bauml, and U. Frese, "Real-time swept volume and distance computation for self collision detection," *Intelligent Robots and Systems*, pp. 1585–1592, 2011.

[15] E. Yoshida and F. Kanehiro, "Reactive robot motion using path replanning and deformation," *2011 IEEE International Conference on Robotics and Automation*, pp. 5456–5462, May 2011.

[16] M. Kallman and M. Mataric, "Motion planning using dynamic roadmaps," *IEEE International Conference on Robotics and Automation, 2004. Proceedings. ICRA '04. 2004*, no. April, pp. 4399–4404 Vol.5, 2004.

[17] B. Marthi, S. J. Russell, J. Wolfe, and C. Sciences, "Angelic Hierarchical Planning: Optimal and Online Algorithms (Revised)," *Electrical Engineering*, 2009.

[18] K. Anderson, N. Sturtevant, R. Holte, and J. Schaeffer, "Coarse-to-Fine Search Techniques, Tech. Rep. April, 2008.

[19] L. Stocco and G. Schrack, "On spatial orders and location codes," in *IEEE Transactions on Computers*, vol. 58, no. 3, 2009, pp. 424–432.

[20] A. Miller and P. Allen, "Graspit! a versatile simulator for robotic grasping," *IEEE Robotics & Automation Magazine*, vol. 11, no. 4, pp. 110–122, 2004.

[21] D. Devaurs, T. Simeon, and J. Cortes, "Parallelizing rrt on distributed-memory architectures," in *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, may 2011, pp. 2261 –2266.

[22] A. Hornung, "3D Collision Avoidance for Navigation in Unstructured Environments, http://www.ros.org/presentations/2011-08-armin.pdf."

[23] M. Tang, D. Manocha, J. Lin, and R. Tong, "Collision-streams: Fast GPU-based collision detection for deformable models," in *I3D '11: Proceedings of the 2011 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, 2011, pp. 63–70.